Factor Graphs and the Sum-Product Algorithm

Frank R. Kschischang*
Brendan J. Frey†
Hans-Andrea Loeliger‡

July 27, 1998

Abstract—A factor graph is a bipartite graph that expresses how a "global" function of many variables factors into a product of "local" functions. Factor graphs subsume many other graphical models including Bayesian networks, Markov random fields, and Tanner graphs. Following one simple computational rule, the sum-product algorithm operates in factor graphs to compute—either exactly or approximately—various marginal functions by distributed message-passing in the graph. A wide variety of algorithms developed in artificial intelligence, signal processing, and digital communications can be derived as specific instances of the sum-product algorithm, including the forward/backward algorithm, the Viterbi algorithm, the iterative "turbo" decoding algorithm, Pearl's belief propagation algorithm for Bayesian networks, the Kalman filter, and certain fast Fourier transform algorithms.

Keywords—Graphical models, factor graphs, Tanner graphs, sum-product algorithm, marginalization, forward/backward algorithm, Viterbi algorithm, iterative decoding, belief propagation, Kalman filtering, fast Fourier transform.

Submitted to *IEEE Transactions on Information Theory*, July, 1998. This paper is available on the web at `http://www.comm.utoronto.ca/frank/factor/`.

*Department of Electrical & Computer Engineering, University of Toronto, Toronto, Ontario M5S 3G4, CANADA (`frank@comm.utoronto.ca`)

†The Beckman Institute, 405 North Mathews Avenue, Urbana, IL 61801, USA (`frey@cs.utoronto.ca`)

‡Endora Tech AG, Gärtenstrasse 120, CH-4052 Basel, SWITZERLAND (`haloeliger@access.ch`)

Typeset with LaTeX at 16:16 on July 27, 1998.

Contents

1 Introduction

1.1 Factor Graphs

A *factor graph* is a bipartite graph that expresses how a "global" function of many variables factors into a product of "local" functions. Suppose, e.g., that some real-valued function $g(x_1, x_2, x_3, x_4, x_5)$ of five variables can be written as the product

$$g(x_1, x_2, x_3, x_4, x_5) = f_A(x_1) f_B(x_2) f_C(x_1, x_2, x_3) f_D(x_3, x_4) f_E(x_3, x_5) \tag{1}$$

of five functions, $f_A, f_B, \ldots, f_E$. The corresponding factor graph is shown in Fig. 1(a). There is a *variable node* for each variable, there is a *function node* for each factor, and the variable node for x_i is connected to the function node for f if and only if x_i is an argument of f.

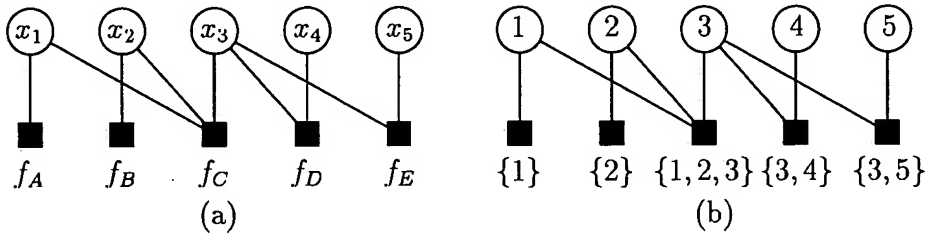

Figure 1: A factor graph that expresses that a global function factors as the product of local functions $f_A(x_1) f_B(x_2) f_C(x_1, x_2, x_3) f_D(x_3, x_4) f_E(x_3, x_5)$. (a) Variable/function view, (b) index/subset view. Variable nodes are shown as circles; function nodes are shown as filled squares.

We will use the following notation. Let $X_S = \{x_i : i \in S\}$ be a collection of variables indexed by a finite set S, where S is linearly ordered by $\leq$. For each $i \in S$, the variable x_i takes on values from some set A_i. Most often S will be a subset of the integers with the usual ordering. If E is a subset of S, then we denote by $X_E = \{x_i : i \in E\}$ to be the subset of variables indexed by E.

A particular assignment of a value to each of the variables of X_S will be referred to as a *configuration* of the variables. Configurations of the variables can be viewed as being elements of the Cartesian product $A_S \triangleq \prod_{i \in S} A_i$, called the *configuration space.* For concreteness, we suppose that the components of configurations are ordered as in S, so that if $S = \{i_1, i_2, \ldots, i_N\}$ with $i_1 \leq i_2 \leq \cdots \leq i_N$, then a typical configuration a is written as $a = (a_{i_1}, a_{i_2}, \ldots, a_{i_N})$ with $a_{i_j} \in A_{i_j}$ for $j = 1, \ldots, N$. Of course, the configuration a is equivalent to the multiple variable assignment $x_{i_1} = a_{i_1}$, $x_{i_2} = a_{i_2}$, $\ldots$, and *vice versa.* By abuse of notation, if a is a particular configuration, we will write $X_S = a$ for this assignment. We will have occasion to view configurations both as assignments of values to variables and as elements of A_S.

We will also have occasion to consider *subconfigurations*: if $E = \{j_1, j_2, \ldots, j_M\} \subset S$, with $j_1 \leq j_2 \leq \cdots \leq j_M$, and a is any configuration, then the M-tuple $a_E = (a_{j_1}, \ldots, a_{j_M})$ is called the subconfiguration of a with respect to E. The set $\{a_E : a \in A_S\}$ of all subconfigurations with respect to E is denoted by A_E; clearly $A_E = \prod_{i \in E} A_i$. Again, by abuse of notation, if a_E is a particular subconfiguration with respect to E, we will write $X_E = a_E$ for the multiple variable assignment $x_{j_1} = a_{j_1}$, $x_{j_2} = a_{j_2}$, etc.

Finally, if $C \subset A_S$ is some set of configurations, we will denote by C_E the set of subconfigurations of the elements of C with respect to E, i.e., $C_E = \{a_E : a \in C\}$. Clearly $C_E \subset A_E$.

Let g: $A_S \to R$ be a function with the elements of X_S as arguments. For the moment, we require the domain R of g to be equipped with a binary product (denoted ' $\cdot$ ') and a unit element (denoted 1) satisfying, for all u, v, and w in R,

$$1 \cdot u = u, \quad u \cdot v = v \cdot u, \quad (u \cdot v) \cdot w = u \cdot (v \cdot w),$$

so that R forms a commutative semigroup with unity. The reader will lose nothing essential in most cases by assuming that R is a field, e.g., the real numbers, under the usual product. We will usually denote the product of elements x and y by the juxtaposition xy, and only occasionally as $x \cdot y$.

Suppose, for some collection Q of subsets of S, that the function g factors as

$$g(X_S) = \prod_{E \in Q} f_E(X_E) \tag{2}$$

where, for each $E \in Q$, f_E: $A_E \to R$ is a function of the subconfigurations with respect to E. We refer to each factor $f_E(X_E)$ in (2) as a *local function*. (If some $E \in Q$ is empty, i.e., $E = \emptyset$, we interpret the corresponding local "function" $f_\emptyset$ with no arguments as a constant in R.) Often, as is common practice in probability theory, we will use an abbreviated notation in which the arguments of a function determine the function domain, so that, e.g., $f(x_1, x_2)$ would denote a function from $A_1 \times A_2 \to R$, as would $f(x_2, x_1)$. In this abbreviated notation, (2) would be written as $g(X_S) = \prod_{E \in Q} f(X_E)$.

A factor graph representation of (2) is a bipartite graph denoted $F(S, Q)$ with vertex set $S \cup Q$ and edge set $\{\{i, E\} : i \in S, E \in Q, i \in E\}$. In words, $F(S, Q)$ contains an edge $\{i, E\}$ if and only if $i \in E$, i.e., if and only if x_i is an argument of the local function f_E. Those vertices that are elements of S are called *variable nodes* and those vertices that are elements of Q are called *function nodes*. For example, in (1), we have $S = \{1, 2, 3, 4, 5\}$ and $Q = \{\{1\}, \{2\}, \{1, 2, 3\}, \{3, 4\}, \{3, 5\}\}$, which gives the factor graph $F(S, Q)$ shown in Fig. 1(b). Throughout, we will translate freely between factor graphs labeled with variables and local functions (the 'variable/local function view' of Fig. 1(a)) and the corresponding factor graph labeled with variable indices and index subsets (the 'index/subset view' of Fig. 1(b)). To avoid the more precise but often quite tedious mention of functions

"corresponding to" function nodes, and variables "corresponding to" variable nodes, we will blur the distinction between the nodes and the objects associated with them, thereby making it legitimate to refer to, say, the arguments of a function node f, or the edges incident on a variable x_i.

It will often be useful to refer to an arbitrary edge of a factor graph. Such an edge $\{v, w\}$ by definition is incident on a function node and a variable node; the latter is called the variable *associated* with the given edge, and is denoted by $x_{\{v,w\}}$.

1.2 Prior Art

We will see in Section 2 that factor graphs subsume many other graphical models in signal processing, probability theory, and coding, including Markov random fields [19, 21, 32], Bayesian networks [20, 31] and Tanner graphs [35, 38, 39]. Our original motivation for introducing factor graphs was to make explicit the commonalities between Bayesian networks (also known as belief networks, causal networks, and influence diagrams) and Tanner graphs, both of which had previously been used to explain the iterative decoding of turbo codes and low-density parity check codes [11, 22, 25, 26, 30, 38, 39]. In that respect, factor graphs and their applications to coding are just a slight reformulation of the approach of Wiberg, *et al.* [39]. However, a main thesis of this paper is that factor graphs may naturally be used in a wide variety of fields other than coding, including signal processing, system theory, expert systems, and artificial neural networks.

It is plausible that many algorithms in these fields are naturally expressed in terms of factor graphs. In this paper we will consider only one such algorithm: the *sum-product algorithm*, which operates in a factor graph by passing "messages" along the edges of the graph, following a single, simple, computational rule. (By way of preview, a very simple example of the operation of the sum-product algorithm operating in the factor graph of Fig. 1 is given in the next subsection.)

The main purpose of this essentially tutorial paper is to illuminate the simplicity of the sum-product algorithm in the general factor graph setting, and then point out a variety of applications. In parallel with the development of this paper, Aji and McEliece [1, 2] develop the closely related "generalized distributive law," an alternative approach based on the properties of junction trees (and not factor graphs). Aji and McEliece also point out the commonalities among a wide variety of algorithms, and furnish an extensive bibliography. Forney [10] gives a nice overview of the development of many of these algorithms, with an emphasis on applications in coding theory.

The first appearance of the sum-product algorithm in the coding theory literature is probably Gallager's decoding algorithm for low-density parity-check codes [14]. The optimum (minimum probability of symbol error) detection algorithm for codes, sometimes referred to as the MAP (maximum *a posteriori* probability) algorithm or the BCJR algo-

rithm (after the authors of [4]) turns out to be a special case of the sum-product algorithm applied to a trellis. This algorithm was developed earlier in the statistics literature [5] and perhaps even earlier in classified work due to L. R. Welch [29]. In the signal processing literature, and particularly in speech processing, this algorithm is widely known as the forward-backward algorithm [33]. Pearl's belief propagation and belief revision algorithms, widely applied in expert systems and in artificial intelligence, turn out to be examples of the sum-product algorithm operating in a Bayesian network; see [31, 20] for textbook treatments. Neural network formulations of factor graphs have also been used for unsupervised learning and density estimation; see, e.g., [11].

In coding theory, Tanner [35] generalized Gallager's bipartite graph approach to low-complexity codes and also developed versions of the sum-product algorithm. Tanner's approach was later generalized to graphs with hidden (state) variables by Wiberg, *et al.* [39, 38]. In coding theory, much of the current interest in so-called "soft-output" decoding algorithms stems from the near-capacity-achieving performance of turbo codes, introduced by Berrou, *et al.* [7]. The turbo decoding algorithm was formulated in a Bayesian network framework by McEliece, *et al.* [30], and Kschischang and Frey [22].

1.3 A Sum-Product Algorithm Example

As we will describe more precisely in Section 4, the sum-product algorithm can be used (in a factor graph that forms a tree) to compute a function *summary* or *marginal.* For example, consider the specific case in which the global function defined in (1) is real-valued and represents the conditional joint probability mass function of a collection of discrete random variables, given some observation y. We might be interested in, say, the marginal function

$$p(x_1|y) = \sum_{x_2}\sum_{x_3}\sum_{x_4}\sum_{x_5} g(x_1, x_2, x_3, x_4, x_5).$$

From the factorization given by (1), we write

$$\begin{aligned} p(x_1|y) &= \sum_{x_2}\sum_{x_3}\sum_{x_4}\sum_{x_5} f_A(x_1) f_B(x_2) f_C(x_1, x_2, x_3) f_D(x_3, x_4) f_E(x_3, x_5) \\ &= f_A(x_1) \underbrace{\sum_{x_2} f_B(x_2) \sum_{x_3} f_C(x_1, x_2, x_3) \underbrace{\underbrace{\sum_{x_4} f_D(x_3, x_4)}_{f_D(x_3,x_4)\downarrow x_3} \underbrace{\sum_{x_5} f_E(x_3, x_5)}_{f_E(x_3,x_5)\downarrow x_3}}_{f_{DE}(x_3,x_4,x_5)\downarrow x_3}}_{f_{BCDE}(x_1,x_2,x_3,x_4,x_5)\downarrow x_1} \end{aligned} \tag{3}$$

where we write $f_{DE}(x_3, x_4, x_5)$ for the product $f_D(x_3, x_4) f_E(x_3, x_5)$, and $f_{BCDE}(x_1, x_2, x_3, x_4, x_5)$ for the product $f_B(x_2) f_C(x_1, x_2, x_3) f_D(x_3, x_4) f_E(x_3, x_5)$.

In (3) we have identified the various factors that need to be computed to obtain $p(x_1|y)$. We have used a notation for a *summary operator* that will be introduced in Section 3. In this example, for $i \in E$, the notation $f(X_E) \downarrow x_i$, called the summary of $f(X_E)$ for x_i, is defined as the marginal function

$$f(X_E) \downarrow x_i = \sum_{x_j : j \in E \setminus \{i\}} f(X_E)$$

obtained by summing over all possible subconfigurations of the arguments of f, *other* than x_i. In this notation, $p(x_1|y) = g(X_S) \downarrow x_1$.

Our primary observation is that $g(X_S) \downarrow x_1$ can be computed knowing just $f_A(x_1)$ and $f_{BCDE}(x_1) \downarrow x_1$. The latter factor can be computed knowing just $f_B(x_2)$, $f_C(x_1, x_2, x_3)$ and $f_{DE}(x_3, x_4, x_5) \downarrow x_3$. In turn, $f_{DE}(x_3, x_4, x_5) \downarrow x_3$ can be computed knowing just $f_D(x_3, x_4) \downarrow x_3$ and $f_E(x_3, x_4) \downarrow x_4$.

These products can be "gathered" in a distributed manner in the factor graph for g, as shown in Fig. 2. Imagine a processor associated with each node of the factor graph, capable of performing local computations (i.e., computing local function products and local function summaries), and imagine also that these processors are capable of communicating with adjacent processors by sending "messages" along the edges of the factor graph. The messages are descriptions of summaries of various local function products. By passing messages as shown in Fig. 2, all of the factors necessary for the computation of $g(X_S) \downarrow x_1$ become available at x_1.

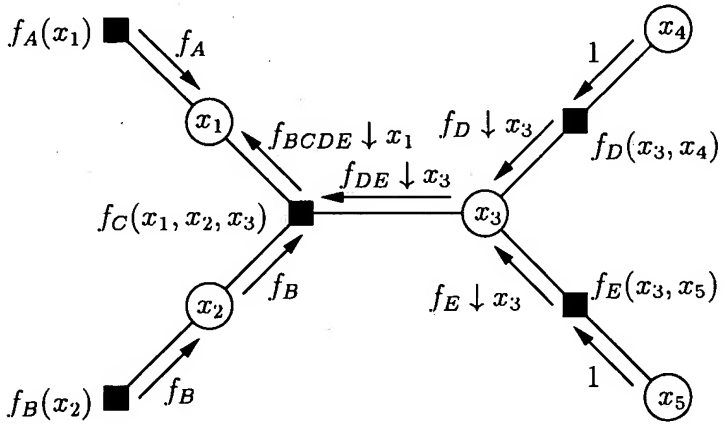

Figure 2: Computing $g(x_1, \ldots, x_5) \downarrow x_1$ by the sum-product algorithm.

As we will make clear later, each processor needs only to follow a single, simple, computational rule: the message passed from node v to node w on the edge $\{v, w\}$ is the product of the messages that arrive on all *other* edges incident on v with the local function at v (if any), summarized for the variable $x_{\{v,w\}}$ associated with the given edge. The reader may verify that precisely this rule was followed in generating the messages

passed in the factor graph of Fig. 2. In a tree, the algorithm terminates by computing at a variable node (x_1, in this case) the product of all incoming messages.

As an exercise, the reader may wish to verify that the various factors needed to compute *each* marginal function can be obtained as products of the messages sent by the sum-product algorithm, as is shown in Fig. 18 of Section 4. From this conceptually simple computational procedure operating in the corresponding factor graph, we will be able to derive the wide variety of algorithms mentioned in the Abstract.

1.4 Notational Preliminaries

We will need the following terminology and ideas from graph theory. Let $G(V, E)$ be a graph with vertex set V and edge set E. Edges are not directed; an edge between two vertices v and w is the unordered pair $e = \{v, w\}$, and e is said to be *incident on* v and w. The *degree* $\partial(v)$ of a vertex v is the number of edges incident on v. For every v, the set $n(v)$ of *neighbors* of v is $n(v) = \{w : \{v, w\} \in E\}$, the set of vertices that share an edge with v. Clearly v has $\partial(v)$ distinct neighbors. If f is a function node in a factor graph, then $X_{n(f)}$ is the set of arguments of f.

A *path* between vertices v and w in G is defined, as usual, as a sequence of distinct edges $\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{L-1}, v_L\}$ in E with $v_1 = v$, and $v_L = w$. A *cycle* in G is a path between a vertex v and itself. If there is a path between v and w, then v and w are said to be *connected* in G. Since vertices are always considered to be connected to themselves, connectedness is an equivalence relation on the vertices of G; the disjoint equivalence classes induce disjoint subgraphs of G called the (connected) *components* of G. If G comprises a single component—as will be the case for the majority of factor graphs considered in this paper—then G is said to be *connected.*

A graph G is a *tree* if it is connected and has no cycles. A graph G is a tree if and only if there is a unique path between any pair of distinct vertices in G. In a tree, a vertex v is said to be a *leaf node* if $\partial(v) \leq 1$. In any finite tree of more than one node, there are always at least two leaf nodes. If u and w are two arbitrary but distinct vertices in a tree G, and v is a leaf node distinct from u and w, then the path from u to w does not pass through v. Since u and w are arbitrary (though distinct from v) this means that if v and the edge incident on v are deleted from G, the resulting graph is still a tree.

More generally, if G is a tree then the graph obtained by cutting (i.e., removing) any edge $\{v, w\}$ from G is the union of two components, one denoted $G_{w \to v}$ (containing v, but not w) and the other denoted $G_{v \to w}$ (containing w but not v), both of which are themselves trees. The notation is intended to be mnemonic: $G_{w \to v}$ is the subgraph of G as "viewed" from the edge $\{v, w\}$ while facing in the direction of v.

Variables corresponding to nodes in distinct components of a factor graph F are said

to *contribute independently* to the corresponding (global) function. More generally, two variable subsets X_E and $X_{E'}$ contribute independently to g if E and E' are contained in distinct components of F. If g is the joint probability mass or density function of a collection of random variables, and if x_i and x_j are variables that contribute independently to g, then the corresponding random variables are independent. (The converse is not necessarily true.)

We will also need the following useful notation called "Iverson's convention" [15, p. 24] for indicating the truth of a logical proposition: if P is a Boolean proposition, then $[P]$ is the binary function that indicates whether or not P is true, i.e.,

$$[P] = \begin{cases} 1 & \text{if } P; \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

We will use Iverson's convention in formulas only in those contexts in which it is sensible to have a $\{0,1\}$-valued quantity. We will occasionally use square brackets simply as brackets, but this should cause no confusion since the enclosed quantity will in those cases not be a Boolean proposition.

We will often have occasion to consider binary indicator functions, i.e., $\{0,1\}$-valued functions of several variables. A convenient graphical representation for a binary indicator function of three variables taking values in finite alphabets is a *trellis section*. If $f(x,y,z)$ is such a function, then there exists a set $B \subset A_x \times A_y \times A_z$ such that $f(x,y,z) = [(x,y,z) \in B]$. We take the set B as defining the set of labeled edges in a directed bipartite graph, called a trellis section. We take the set A_x as the set of "left vertices," the set A_z as the set of "right vertices," and the set A_y as the set of "edge labels." Each triple $(x,y,z) \in B$ defines an edge with left vertex x, right vertex z, and label y. For example, for binary variables x, y, and z, (considered as elements of the field $GF(2)$) the trellis sections corresponding to $[x+y=z]$ and $[xy=z]$ are shown in Fig. 3(a) and (b), respectively. In this figure, as in all of our figures of trellis sections, we have placed the left vertices on the left and the right vertices on the right, so that an arrow indicating the orientation of an edge is not required.

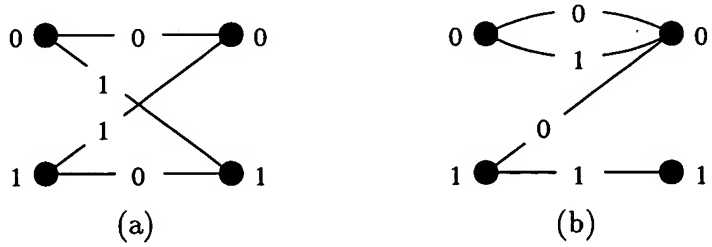

Figure 3: Trellis sections corresponding to binary indicator functions of three variables: (a) $[x+y=z]$, (b) $[xy=z]$.

1.5 Organization of the Paper

The remainder of this paper is organized as follows. In Section 2, to illustrate the broad applicability of factor graphs, and to motivate the remainder of the paper, we provide a number of examples of factor graphs in a variety of areas including coding theory, systems theory, probability theory, neural networks and signal processing.

In Section 3, we study cross-sections and projections of functions, and formulate the general notion of a "summary operator" that acts on a function projection to create marginal functions.

The operation of the sum-product algorithm is described in Section 4. As we have already briefly described, the sum-product algorithm operates using a local computational procedure that is characterized by one conceptually simple computational rule. In a tree, we prove that this procedure results in exact function marginalization. (In Appendix B we provide a complexity analysis for the important case in which the summary operator is defined in terms of a sum operation like that in the example of Section 1.3.)

In Section 5 we apply the sum-product algorithm to the factor graphs of Section 2, and obtain a variety of well-known algorithms as special cases. These include the forward/backward algorithm, the Viterbi algorithm, Pearl's belief propagation algorithms for Bayesian networks, and the Kalman filter.

In Section 6, we describe some of the possible transformations that may be applied to a factor graph without changing the function that it represents. This will be used to motivate a procedure for exact marginalization in factor graphs with cycles. As an application of this procedure, we derive a Fast Fourier Transform algorithm as a special case of the sum-product algorithm.

Some concluding remarks are offered in Section 7.

2 Examples of Factor Graphs

Having defined the general concept of factor graphs, we now give some examples of the way in which factor graphs may be used to represent useful functions. In Section 5, we will describe some of the applications of the sum-product algorithm using these graphs.

Among all multi-variable functions that we might wish to represent by a factor graph, two particular classes stand out: set membership *indicator functions,* whose value is either 0 or 1, and *probability distributions.* Such functions are often interpreted as models—set theoretic or probabilistic, respectively—of a physical system. For example, Willems' system theory [40] starts from the view that a "system" (i.e., a model) *is* simply a set

of allowed trajectories in some configuration space. Factorizations of such functions can give important structural information about the model. Moreover, the structure of the factor graph has a strong influence on the performance of the sum-product algorithm; e.g., we will see in Section 4 that the algorithm is "exact," in a well-defined sense, only if the graph has no cycles. We shall see below that a number of established modeling styles, both set theoretic and probabilistic, correspond to factor graphs with a particular structure.

2.1 Indicating Set Membership: Tanner Graphs

We start with set membership indicator functions. As usual let S be an index set, and let A_S denote a configuration space. In many applications, particularly in coding theory, we work with a fixed subset C of A_S, which we think of as the set of codewords or *valid configurations*. In these applications, we will be interested in the set membership indicator function

$$g(X_S)\colon A_S \to \{0,1\}$$

defined, for all $a \in A_S$, by

$$g(a) = [a \in C] = \begin{cases} 1 & \text{if } a \in C; \\ 0 & \text{otherwise.} \end{cases}$$

Of course, in this paper, we will be interested in situations in which $g(X_S)$ factors as in (2). In particular, suppose that each factor, $f_E(X_E)$, is itself a binary indicator function, that indicates whether a particular subconfiguration a_E is "locally valid." Both the global function and all local functions take values in the set $\{0,1\}$, considered as a subset of the reals under the usual multiplication. In this setup, a configuration $a \in A_S$ is valid (i.e., $g(a) = 1$) if and only if $f_E(a_E) = 1$ for all $E \in Q$. The product acts as a logical conjunction (AND) operator: a (global) configuration is valid if and only if all of its subconfigurations are valid.

In this context, local functions are often referred to as (local) *checks*, and the corresponding function nodes in the factor graph are also called *check nodes*. Given a code $C \subset A_S$, we will often (somewhat loosely) refer to a factor graph representation for C; what we strictly mean in such situations is a factor graph representation for the indicator function $g(a) = [a \in C]$. We will often refer to a factor graph for a set membership indicator function that factors as a product of local checks as a *Tanner graph*.

Example 1. (*Linear codes*)
Of course, every code has a factor graph representation (and in general more than one). A

convenient way to construct a factor graph for a *linear* code is to start with a parity-check matrix for the code.

To illustrate, consider the linear code C over $GF(2)$, defined by the parity check matrix

$$H = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}, \tag{5}$$

consisting of of all binary 6-tuples $\mathbf{x} \triangleq (x_1, x_2, \ldots, x_6)$ satisfying $H\mathbf{x}^T = 0$. Since every linear code has a parity-check matrix, this approach applies to all linear codes.

In effect, each row of the parity-check matrix gives us an equation that must be satisfied by $\mathbf{x}$, and $\mathbf{x} \in C$ if and only if *all* equations are satisfied. Thus, if a binary function indicating satisfaction of each equation is introduced, the product of these functions indicates membership in the code.

In our example, H has three rows, and hence the code membership indicator function $\mu(x_1, x_2, \ldots, x_6)$ can be written as a product of three local indicator functions:

$$\begin{aligned} \mu(x_1, x_2, \ldots, x_6) &= [(x_1, x_2, \ldots, x_6) \in C] \\ &= [x_1 \ominus x_2 \ominus x_5 = 0][x_2 \ominus x_3 \ominus x_6 = 0][x_1 \ominus x_3 \ominus x_4 = 0], \end{aligned}$$

where we have again used Iverson's convention and where $\ominus$ denotes the sum in $GF(2)$. The corresponding factor graph is shown in Fig. 4.

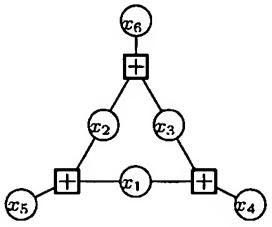

Figure 4: A factor graph for the binary linear code of Example 1.

In Fig. 4, we have used a special symbol for the parity checks (a square with a "+" sign instead of a black square). In fact, we will freely use a variety of symbols for function nodes, depending on the type of local function. Variable nodes will always be drawn as circles; double circles (as in Fig. 6, described below) will sometimes be used to indicate auxiliary variables (states).

Example 2. (*Logic circuits*)
Many readers may be surprised to note that they are already quite familiar with certain

factor graphs, for example, the factor graph shown in Fig. 5. Here, the local checks are drawn as logic gates, to remind us of the definition of the corresponding binary indicator function. For example, the AND gate with inputs u_1 and u_2 and output x_1 represents the binary indicator function $f(u_1, u_2, x_1) = [x_1 = u_1 \text{ AND } u_2]$.

Viewed as a factor graph, the logic circuit of Fig. 5 represents the global function

$$g(u_1, u_2, u_3, u_4, x_1, x_2, y) = [x_1 = u_1 \text{ AND } u_2][x_2 = u_3 \text{ AND } u_4][y = x_1 \text{ OR } x_2]. \quad (6)$$

The function g takes on the value 1 if and only if its arguments form a configuration consistent with the correct functioning of the logic circuit.

In general, any logic circuit can be viewed as a factor graph. The local function corresponding to some elementary circuit takes on the value 1 if and only if the corresponding variables behave (locally) as required by the circuit. If necessary, auxiliary variables (not directly observable as input or outputs) like x_1 and x_2 in Fig. 5 may be introduced between logic gates. As we shall see in the next example, the introduction of such auxiliary variables can give rise to particularly "nice" representations of set-membership indicator functions.

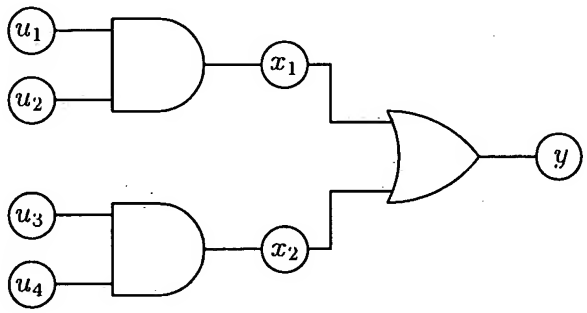

Figure 5: A logic circuit—also a factor graph!

Example 3. (*Auxiliary variables*)
When dealing with binary functions indicating membership in some set $C \subset A_S$ of valid configurations, it will often be useful to introduce *auxiliary* or *state* or *hidden* variables. By this, we mean the introduction of a set $T \supset S$, and a set $X_{T \setminus S}$ of variables indexed by T but not by S, called auxiliary variables.

In this setup, we view A_S as being the set of subconfigurations with respect to S of the enlarged configuration space A_T. We can then introduce a set $D \subset A_T$ of configurations in the enlarged space. Provided that $D_S = C$, i.e., that the subconfigurations of the elements of D with respect to S is equal to C, we will consider a factor graph for D to be a valid factor graph for C. Following Forney [10] we will sometimes refer to such a factor graph—and any factor graph for a set membership indicator function having auxiliary

variables—as as a TWL (Tanner/Wiberg/Loeliger) graph for C. As mentioned earlier, auxiliary variable nodes are indicated with a double circle in our factor graph diagrams.

To illustrate this idea, Fig. 6(b) shows a TWL graph for the binary code of Example 1. In addition to the variable nodes $x_1, x_2, \ldots, x_6$, there are also variable nodes for the auxiliary variables $s_0, s_1, \ldots, s_6$. The definition of the local checks, which are drawn as generic function nodes (black squares), is given in terms of a trellis for the code, which is shown in Fig. 6(a).

A trellis for C is defined by the property that the sequence of edge labels encountered in each directed path (left to right) from the leftmost vertex to the rightmost vertex in the trellis is always a codeword in C, and that each codeword is represented by at least one such path.

Here, the auxiliary variables $s_0, \ldots, s_6$ correspond to the trellis states, and each local check represents one trellis section, i.e., the ith local function (counting from the left) indicates which triples (s_{i-1}, x_i, s_i) are valid (state, output, next state) transitions in the trellis, drawn according to our convention for indicator functions of three variables introduced in Section 1.

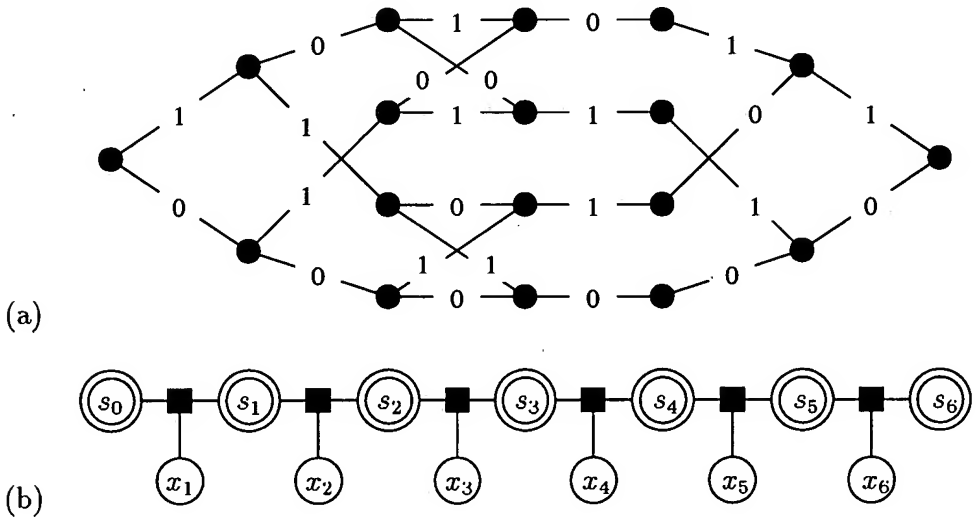

Figure 6: A trellis (a) and the corresponding TWL graph (b) for the code of Fig. 4.

In this example, the second trellis section from the left in Fig. 6 consists of the following triples (s_1, x_2, s_2):

$$S_2 = \{(0,0,0), (0,1,2), (1,1,1), (1,0,3)\}, \tag{7}$$

where the alphabet of the state variables s_1 and s_2 was taken to be $\{0,1\}$ and $\{0,1,2,3\}$, respectively, numbered from bottom to top in Fig. 6(a). The corresponding check (i.e., local function) in the TWL graph is the indicator function $f(s_1, x_2, s_2) = [(s_1, x_2, s_2) \in S_2]$.

The method described in this example to obtain a factor graph from a trellis is completely general and applies to any trellis. Since every code can be represented by a trellis (see [36] for a recent survey of results in the theory of the trellis structure of codes), this shows that a cycle-free factor graph exists for every code (in fact, for every set membership function).

In general, the purpose of introducing auxiliary variables (states) is to obtain "nice" factorizations of the global function that are not otherwise possible.

Example 4. (*State-space models*)
Trellises are convenient representations for a variety of signal models. For example, the generic factor graph of Fig. 7 can represent any time-invariant (or indeed, time-varying) state space model. As in Fig. 6, each local check represents a trellis section, an indicator function for the set of allowed combinations of left state, input symbol, output symbol, and right state. (Here, a trellis edge has two labels.)

For example, the classical linear state space model is given by the equations

$$\begin{array}{rcl} x(j+1) & = & Ax(j) + Bu(j), \\ y(j) & = & Cx(j) + Du(j), \end{array} \tag{8}$$

where $j \in \mathbb{Z}$ is the discrete time index, where $u(j) = [u_1(j), \ldots, u_k(j)]$ are the time-j input variables, $y(j) = [y_1(j), \ldots, y_n(j)]$ are the output variables, $x(j) = [x_1(j), \ldots, x_m(j)]$ are the state variables, and where A, B, C, and D are matrices of the appropriate dimensions. The equation is over some (finite or infinite) field F.

Any such system gives rise to the factor graph of Fig. 7. The time-j check function $f(x(j), u(j), y(j), x(j+1)) \colon F^m \times F^k \times F^n \times F^m \to \{0,1\}$ is

$$f(x(j), u(j), y(j), x(j+1)) = [x(j+1) = Ax(j) + Bu(j)][y(j) = Cx(j) + Du(j)].$$

In other words, the check function enforces the local behavior required by (8).

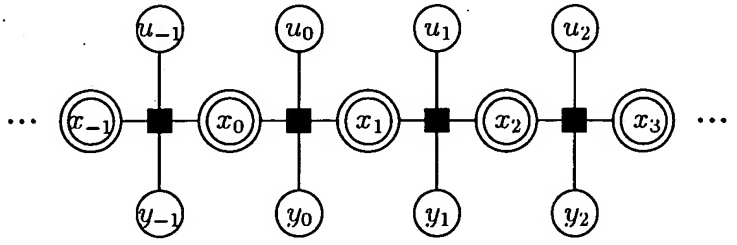

Figure 7: Generic factor graph for a time-invariant trellis.

2.2 Probability Distributions

We turn now to another important class of functions that we will represent by factor graphs: probability distributions. Since conditional and unconditional independence of random variables is expressed in terms of a factorization of their joint probability mass or density function, factor graphs for probability distributions arise in many situations. We expose one of our primary application interests by starting with an example from coding theory.

Example 5. (*Decoding*)
Consider the situation most often modeled in coding theory, in which a codeword $(x_1, \ldots, x_n)$ is selected with uniform probability from a code C of length n, and transmitted over a discrete memoryless channel with corresponding output $(y_1, \ldots, y_n)$. Since the channel is memoryless, by definition the conditional probability mass or density function evaluated at a particular channel output assumes the product form:

$$f(y_1, \ldots, y_n | x_1, \ldots, x_n) = \prod_{i=1}^{n} f(y_i | x_i).$$

The *a priori* probability of selecting a particular codeword is a constant, and hence the *a priori* joint probability mass function for the codeword symbols is proportional to the code set membership indicator function. It follows that the joint probability mass function of $\{x_1, \ldots, x_n, y_1, \ldots, y_n\}$ is proportional to

$$f(x_1, \ldots, x_n, y_1, \ldots, y_n) = [(x_1, \ldots, x_n) \in C] \prod_{i=1}^{n} f(y_i | x_i). \tag{9}$$

Of course, as described in the previous subsection, the code membership indicator function itself may factor into a product of local indicator functions. For example if C is the binary linear code of Example 1, we have

$$f(x_1, \ldots, x_6, y_1, \ldots, y_6) = [x_1 \ominus x_2 \ominus x_5 = 0] \cdot [x_2 \ominus x_3 \ominus x_6 = 0] \cdot [x_1 \ominus x_3 \ominus x_4 = 0] \cdot \prod_{i=1}^{6} f(y_i | x_i),$$

whose factor graph is shown in Fig. 8(a). We see that a factor graph for the joint probability mass function of codeword symbols and channel output symbols is obtained simply by augmenting the factor graph for the code itself.

In decoding, we invariably work with the conditional joint probability mass function for the codeword symbols given the observation of the channel output symbols. As will be discussed in more detail in Section 3.1, in general function terms we deal with a (scaled)

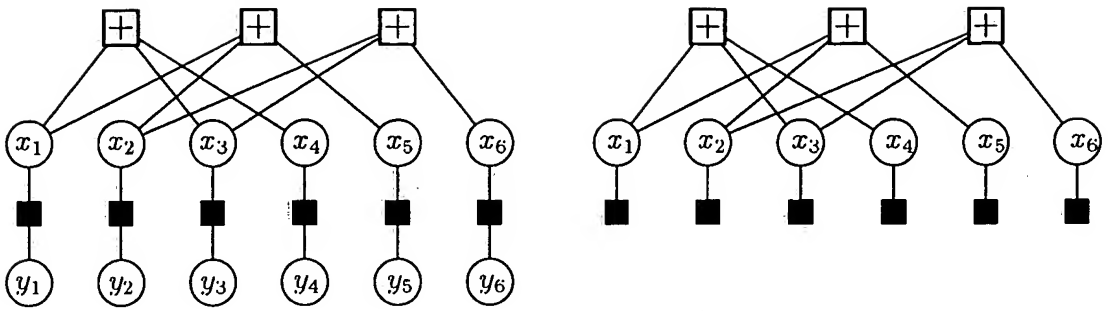

Figure 8: Factor graphs for (a) the joint probability density function of channel input and output for the binary linear code of Fig. 4, (b) the cross-section after observation of a particular channel output vector.

cross-section of the joint probability mass function defined in (9). It turns out that a factor graph for a function cross-section is obtained from a factor graph for the function simply by removing the nodes corresponding to the observed variables, and replacing all local functions with their cross-sections; this is shown for our example code in Fig. 8(b). The cross-section of the function $f(y_i|x_i)$ can be interpreted as a function of x_i with *parameter* y_i.

Example 6. (*Markov chains, hidden Markov models, and factor graphs with arrows*) In general, let $f(x_1, \ldots, x_n)$ denote the joint probability mass function of a collection of random variables. By the chain rule of conditional probability, we may always express this function as

$$f(x_1, \ldots, x_n) = \prod_{i=1}^{n} f(x_i|x_1, \ldots, x_{i-1}).$$

For example if $n = 4$, we have

$$f(x_1, \ldots, x_4) = f(x_1)f(x_2|x_1)f(x_3|x_1, x_2)f(x_4|x_1, x_2, x_3)$$

which has the factor graph representation shown in Fig. 9(b).

Because of this chain rule factorization, in situations involving probability distributions, we will often have local functions of the form $f(x_i|\mathrm{a}(x_i))$, where $\mathrm{a}(x_i)$ is some collection of variables referred to as the *parents* of x_i. In this case, motivated by a similar convention in Bayesian networks [20, 31] (see example 8, below), we will sometimes indicate the *child*, i.e., x_i, in this relationship by placing an arrow on the edge leading from the local function f to x_i. We have followed this *arrow convention* in Fig. 9(b)–(d), and elsewhere in this paper.

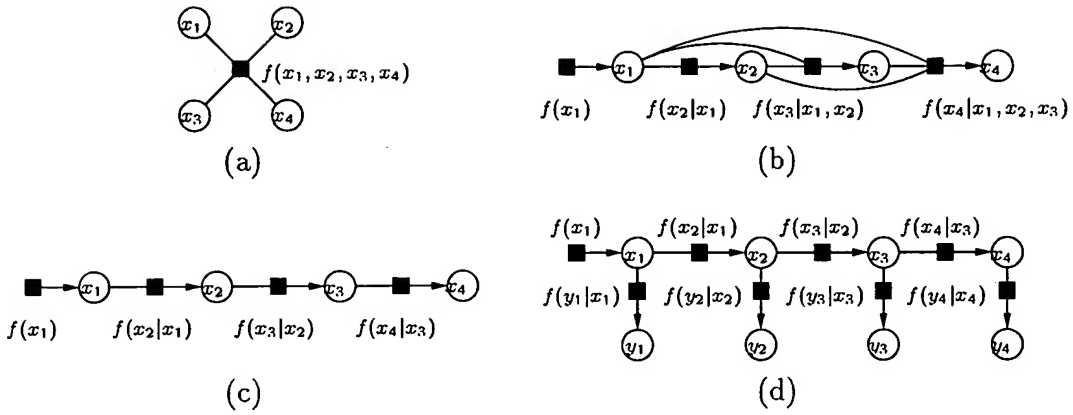

Figure 9: Factor graphs for probability distributions: (a) the trivial factor graph, (b) the chain-rule factorization, (c) a Markov chain, (d) a hidden Markov model.

In general, since all variables appear as arguments of $f(x_n|x_1, \ldots, x_{n-1})$, the factor graph of Fig. 9(b) has no advantage over the trivial factor graph shown in Fig. 9(a). On the other hand, suppose that random variables $\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n$ (in that order) form a Markov chain. We then obtain the nontrivial factorization

$$f(x_1, \ldots, x_n) = \prod_{i=1}^{n} f(x_i|x_{i-1})$$

whose factor graph is shown in Fig. 9(c).

If, in this Markov chain example, we cannot observe each $\mathbf{X}_i$ directly, but instead can observe only the output $\mathbf{Y}_i$ of a memoryless channel with $\mathbf{X}_i$ as input, we obtain a so-called "hidden Markov model." The joint probability mass or density function for these random variables then factors as

$$f(x_1, \ldots, x_n, y_1, \ldots, y_n) = \prod_{i=1}^{n} f(x_i|x_{i-1}) f(y_i|x_i)$$

whose factor graph is shown in Fig. 9(d). Hidden Markov models are widely used in a variety of applications; see, e.g., [33] for a tutorial emphasizing applications in signal processing.

The strong resemblance between the factor graphs of Fig. 9(c) and (d) and the factor graphs representing trellises (Figs. 6(b) and 7) is not accidental; trellises can be viewed as Markov models for codes.

Of course, factor graphs are not the first graph-based language for describing probability distributions. In the next two examples, we describe very briefly the close relationship between factor graphs and models based on undirected graphs (Markov random fields) and models based on directed acyclic graphs (Bayesian networks).

Example 7. (*Markov random fields*)
A Markov random field (see, e.g., [21]) is a graphical model based on an undirected graph $G = (V, E)$ in which each node corresponds to a random variable. The graph G is a *Markov random field* (MRF) if the distribution $p(v_1, \ldots, v_n)$ satisfies the local Markov property:

$$(\forall v \in V) \;\; p(v|V\backslash\{v\}) = p(v|n(v)), \tag{10}$$

where, as usual, $n(v)$ denotes the set of neighbors of v. In other words, G is an MRF if every variable v is independent of non-neighboring variables in the graph, given the values of its immediate neighbors. MRFs are well developed in statistics, and have been used in a variety of applications (see, e.g., [21, 32, 19, 18]). Kschischang and Frey [22] give a brief discussion of the use of MRFs to describe codes.

Recall that a *clique* in a graph is a collection of vertices which are all pairwise neighbors. Under fairly general conditions (e.g., positivity of the joint probability density is sufficient), the joint probability mass function of an MRF can be expressed as the product of a collection of Gibbs potential functions, defined on the set Q of cliques in the MRF. (Indeed, some authors takes this as the defining property of an MRF.) What this means is that the distribution factors as

$$p(v_1, v_2, \ldots, v_N) = Z^{-1} \prod_{E \in Q} f_E(V_E) \tag{11}$$

where Z^{-1} is a normalizing constant, and each $E \in Q$ is a clique. For example (*cf.* Fig. 1), the MRF in Fig. 10(a) can be used to express the factorization

$$p(v_1, v_2, v_3, v_4, v_5) = Z^{-1} f_C(v_1, v_2, v_3) f_D(v_3, v_4) f_E(v_4, v_5).$$

(Although there are other cliques in this graph, e.g., $\{v_1, v_2\}$, we assume that any additional factors are absorbed in one of the factors given; for example, a factor of $f(v_1, v_2)$ would be absorbed by $f_C(v_1, v_2, v_3)$.)

Clearly (11) has precisely the structure needed for a factor graph representation. Indeed, a factor graph representation may be preferable to an MRF in expressing such a factorization, since distinct factorizations, i.e., factorizations with different Qs in (11), may yield precisely the *same* underlying MRF graph, whereas they will always yield distinct factor graphs. (An example in a coding context of this MRF ambiguity is given in [22].)

In the opposite direction, a factor graph F that represents a joint probability distribution can be converted to a Markov random field via a component of the second higher power graph F^2 [22].

Let $F(S, Q)$ be a factor graph, and let F^2 be the graph with the same vertex set as F, with an edge between two vertices v and v' in F^2 if and only if there is there is a path

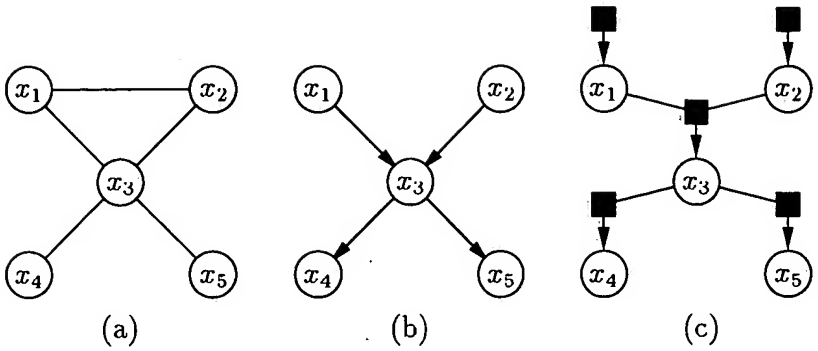

Figure 10: Graphical probability models: (a) a Markov random field, (b) a Bayesian network, (c) a factor graph.

of length two from v to v' in F. Since F is bipartite, F^2 decomposes into at least two components: one component, F_S^2, involving only variable nodes, and the other involving only function nodes. Since the arguments of a function node in F are connected by a path of length two in F, these arguments form a clique in F^2. For example, Fig. 10(a) shows F_S^2 for the factor graph F of Fig. 1.

The following theorem is proved in Appendix A.

Theorem 1 *If $F(S, Q)$ is a factor graph that represents a probability distribution as a product of non-negative factors, then F_S^2 is a Markov random field.*

This theorem can be interpreted as saying that, in a sense, a factor graph is the "square root" of a Markov random field.

Example 8. (*Bayesian networks*)
Bayesian networks (see, e.g., [31, 20, 11]) are graphical models for a collection of random variables that are based on directed acyclic graphs (DAGs). Bayesian networks, combined with Pearl's "belief propagation algorithm" [31], have become an important tool in expert systems over the past decade. The first to connect Bayesian networks and belief propagation with applications in coding theory were MacKay and Neal [25], who independently re-discovered Gallager's earlier work on low-density parity-check codes [14] (including Gallager's decoding algorithm). More recently, at least two papers [22, 30] develop a view of the "turbo decoding" algorithm [7] as an instance of probability propagation in a Bayesian network code model.

Each node v in a Bayesian network is associated with a random variable. Denoting by $\mathsf{a}(v)$ the set of *parents* of v (i.e., the set of vertices *from* which an edge is incident on

v), the distribution represented by the Bayesian network assumes the form

$$p(v_1, v_2, \ldots, v_n) = \prod_{i=1}^{n} p(v_i|\mathbf{a}(v_i)), \tag{12}$$

where, if $\mathbf{a}(v_i) = \emptyset$, (i.e., v_i has no parents) then we take $p(v_i|\emptyset) = p(v_i)$. For example—*cf.* (1)—Fig. 10(b) shows a Bayesian network that expresses the factorization

$$p(v_1, v_2, v_3, v_4, v_5) = p(v_1)p(v_2)p(v_3|v_1, v_2)p(v_4|v_3)p(v_5|v_3). \tag{13}$$

Again, as do Markov random fields, Bayesian networks express a factorization of a joint probability distribution that is suitable for representation by a factor graph. The factor graph corresponding to (13) is shown in Fig. 10(c); *cf.* Fig. 1.

The arrows in a Bayesian network are often useful in modeling the "flow of causality" in practical situations; see, e.g., [31]. Provided that it is not required that a child variable take on some particular value, it is straightforward to *simulate* a Bayesian network, i.e., draw configurations of the variables consistent with the represented distribution. Starting from the variables having no parents, once values have been assigned to the parents of a particular variable x_i, one simply randomly assigns a value to x_i according the (local) conditional probability distribution $p(x_i|\mathbf{a}(x_i))$. Our arrow convention for factor graphs, noted earlier, and illustrated in Fig. 10(c), allows us to retain this advantage of Bayesian networks.

We note that factor graphs are more general than either Markov random fields or Bayesian networks, since they can be used to describe functions that are not necessarily probability distributions. Furthermore, factor graphs have the stronger property that every Markov random field or Bayesian network can be redrawn as a factor graph without information loss, while the converse is not true.

Example 9. (*Logic circuits revisited*)
Probability models are often obtained as an extension of behavioral models. One such case was given in Example 5. For another example, consider the logic circuit of Fig. 5, and suppose that $\{u_1, \ldots, u_4\}$ are random variables that assume particular configurations according to some *a priori* probability distribution. It is not difficult to see that, e.g., the joint probability mass function of u_1, u_2, and x_1 is given by

$$f(u_1, u_2, x_1) = f_{1,2}(u_1, u_2)[x_1 = u_1 \text{ AND } u_2]$$

where $f_{1,2}(u_1, u_2)$ is the joint probability mass function for u_1 and u_2. In this example, we will have

$$f(u_1, u_2, u_3, u_4, x_1, x_2, y) = f_{1,2,3,4}(u_1, u_2, u_3, u_4)g(u_1, u_2, u_3, u_4, x_1, x_2, y)$$

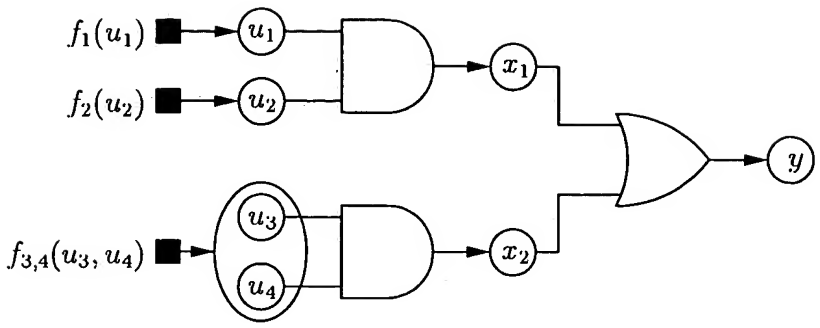

Figure 11: A factor graph representation for the joint probability mass function of the variables in the logic circuit example.

where $f_{1,2,3,4}(\cdot)$ is the joint probability mass function for the u_is and $g(\cdot)$ is the indicator function defined in (6). The factor graph corresponding to the case where $f(u_1, u_2, u_3, u_4)$ itself factors as $f_1(u_1)f_2(u_2)f_{3,4}(u_3, u_4)$ is shown in Fig. 11.

Fig. 11 includes arrows, showing the "flow of causality" from the inputs to the outputs of the logic gates. The pair of input variables u_3, u_4 are "clustered," i.e., treated as a single variable. (Variable clustering and other transformations of factor graphs is discussed in Section 6.) In general, for subsystems with a deterministic input/output relationship among variables, an output variable x_i can be viewed as a child variable having the input variables as parents; for each configuration of the parents, the corresponding conditional probability distribution assigns unit mass to the configuration for x_i consistent with the given input/output relationship.

As discussed, the arrows in Fig. 11 are useful in simulations of the distribution represented by the factor graph, as it is relatively straightforward to go from "inputs" to "outputs" through the graph. As we shall see, the sum-product algorithm will be useful for reasoning in the opposite direction, i.e., computing, say, conditional probability mass functions for the system inputs (or hidden variables) given the observed system output. For example, given models for "faults" in the various subsystems, we may be able to use the sum-product algorithm to reason about the probable cause of an observed faulty output.

2.3 Further Examples

We now give a number of further examples of factor graphs that might be used in a variety of fields, including artificial intelligence, neural networks, signal processing, optimization, and coding theory.

Example 10. (*Computer vision and neural network models*)
Graphical models have found an impressive place in the field of neural network models of perception [18, 17, 9, 12]. (See [11] for a textbook treatment.) Traditional artificial neural networks called "multilayer perceptrons" [34] treat perceptual inference as a function approximation problem. For example, the perceptron's objective might be to predict the relative shift between two images, providing a way to estimate depth. Initially, the perceptron's parameters are set to random values and the perceptron is very bad at predicting depth. Given a set of training images that are *labeled* by depth values, the perceptron's parameters can be estimated so that it can predict depth from a pair of images.

In the more realistic "unsupervised learning" situation, depth labels are not provided, but the neural network is supposed to extract useful structure from the data. One fruitful approach is to design algorithms that learn efficient *source codes* for the data. The parameters of a probability model are adjusted so as to minimize the relative entropy between the empirical data distribution and the distribution given by the model.

The graphical model (factor graph) framework provides a way to specify neurally plausible probability models. In a neural network factor graph, we can associate one local function with each variable such that the local function gives the probability of the variable given the activities of its local input variables. For example, the probability that a binary variable x_i is 1, given its binary inputs $\{x_j : j \in I_i\}$ may take the form,

$$P(x_i = 1 | \{x_j : j \in I_i\}) = 1/(1 + \exp[-\textstyle\sum_{j \in I_i} w_{ij} x_j]),$$

where w_{ij} is the weight on the edge connecting x_i and x_j.

In the simplest case, we create one variable for each input variable. However, we can also introduce unobserved "hidden" or "latent" variables into the graphical model. Once the model is trained so that the marginal distribution over the visible variables is close to the empirical data distribution, we hope the hidden variables will represent useful features.

Although the details of these models and learning algorithms fall beyond the scope of this paper, we present here a brief example. See [13] for the details of a model and a learning algorithm for real-valued variables. In this example, we discuss a binary version of this problem for clarity. Each image in the pair is one-dimensional and contains 6 binary pixels. The training data is generated by randomly drawing pixel patterns for the first image and then shifting the image one pixel to the right or left (with equal probability) to produce the second image. Fig. 12(b) shows 4 examples of these image pairs, with the pair of images placed to highlight the relative shift.

Fig. 12(a) shows the factor graph structure that can be *learned* from examples of these image pairs. Initially, each variable was connected to all variables in adjacent layers and the parameters were set to random values. After learning, the weights associated with